Dear CS124 Grader,

About the Dataset and City Class

This dataset is a collection of the largest cities in the US by population. Each city by default is listed in order from the greatest population to the least, however for this project the cities were sorted in alphabetical order so that the population would not be in order from the first city to the last. This data was collected in 2013, and is one of the more recent data sets available already in the form of a .txt or a .csv file. This data is also very interesting. Cities are constantly growing and shrinking at the same time, especially these large cities. Each entry has 5 attributes, including the city name (string), the state of the city (string), the population of the city (integer), the rank of the city in terms of population (integer), and the percent growth in terms of population from 2000 to 2013 (double). Each of these attributes are fields of my City class.

There are getters and setters because there is no sensitive data. There is no input checking, because all data is perfectly sorted into a combination of strings, integers, and doubles.

The << operator has been overloaded, so each city can be listed in a consistent way in the console. The >, <, >=, <=, and == operators have also been overloaded so a City can be compared to another City.

Sorting Efficiency

Five total sorts were conducted on the same scrambled dataset of varying sizes. The first sorts were conducted with a dataset of 100 City objects, and then 200, all the way up to the full-size set of 1000 City objects. The sorting algorithms used were bubble sort, insertion sort, merge sort, heap sort, and finally a 2-sort step with two algorithms; first insertion sort, and then bubble sort by the name attribute of the City object.

Bubble Sort

Code Source:

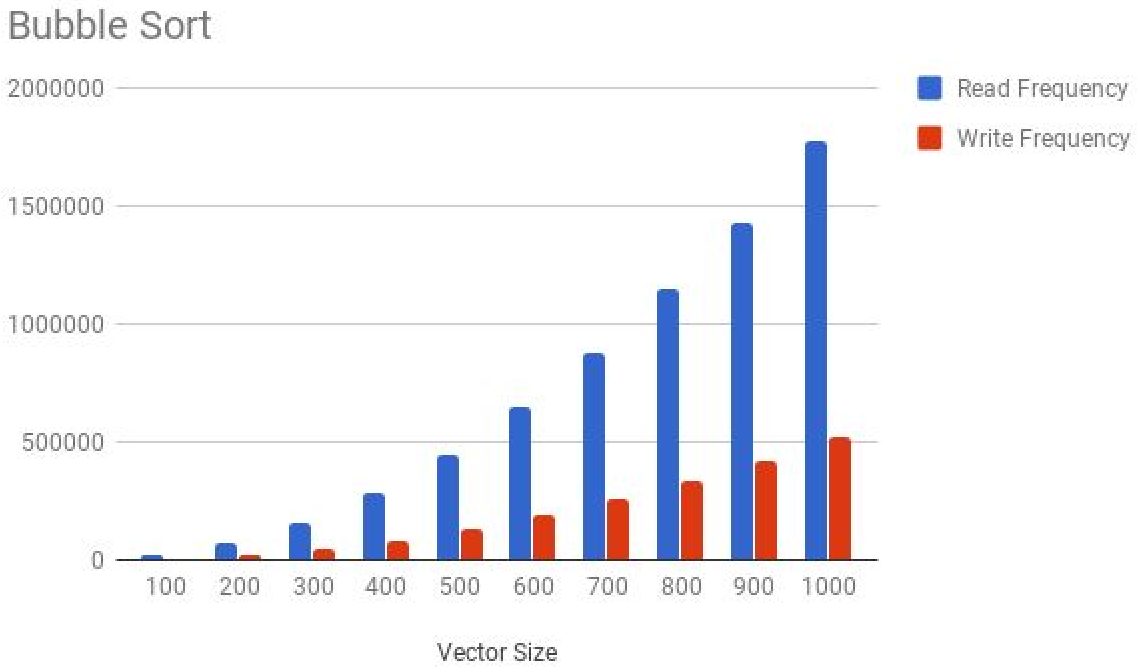
Dion, Lisa. “Main.cpp [Sorting Algorithms].” BlackBoard UVM, BlackBoard, 28 Feb. 2018,bb.uvm.edu/webapps/blackboard/execute/content/file?cmd=view&content_id=_2502870_1&course_id=_117709_1&framesetWrapped=true.

Time Complexity and Efficiency

The time complexity of bubble sort is O(n^2) at worst-case. Below is a table of read and write frequencies recorded from sorting vectors of various sizes.

Vector Size	Read Frequency	Write Frequency

100	17383	5170
200	68393	19642
300	153828	42772
400	280233	80442
500	445130	131988
600	648157	193298
700	876880	258640
800	1145333	337650
900	1430664	415696
1000	1775355	517850



Insertion Sort

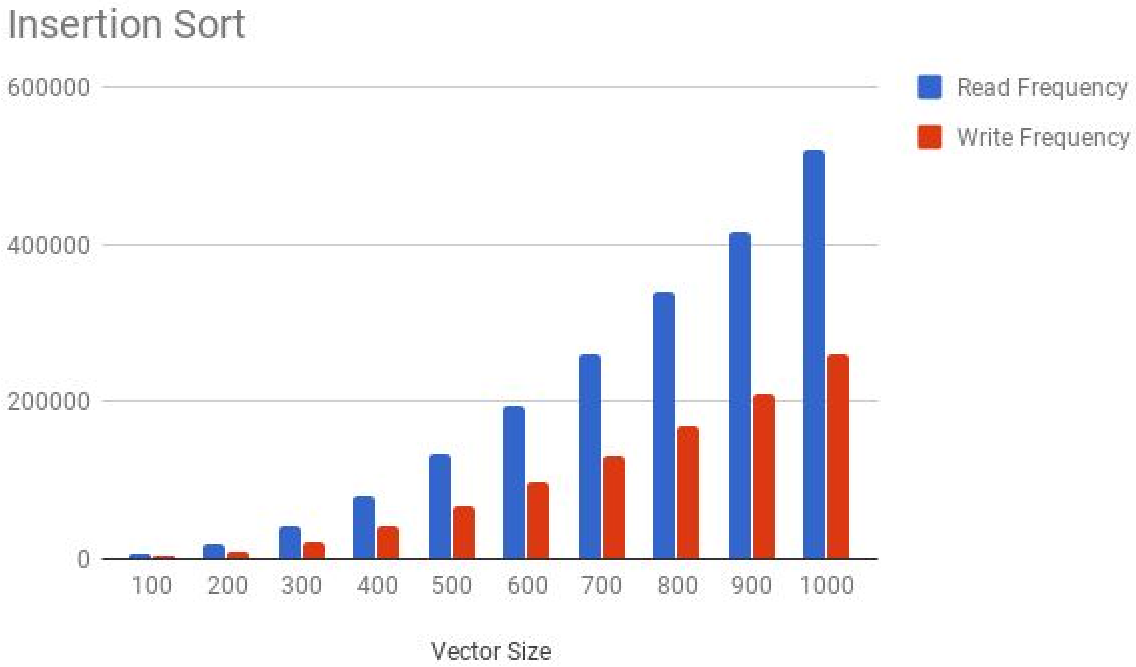
Code Source:

Dion, Lisa. “Main.cpp [Sorting Algorithms].” BlackBoard UVM, BlackBoard, 28 Feb. 2018,bb.uvm.edu/webapps/blackboard/execute/content/file?cmd=view&content_id=_2502870_1&course_id=_117709_1&framesetWrapped=true.

Time Complexity and Efficiency

The time complexity of insertion sort is O(n^2) at worst-case. Below is a table of read and write frequencies recorded from sorting vectors of various sizes.

Vector Size	Read Frequency	Write Frequency
100	5269	2684
200	19841	10020
300	43071	21685
400	80841	40620
500	132487	66493
600	193897	97248
700	259339	130019
800	338449	169624
900	416595	208747
1000	518849	259924



Merge Sort

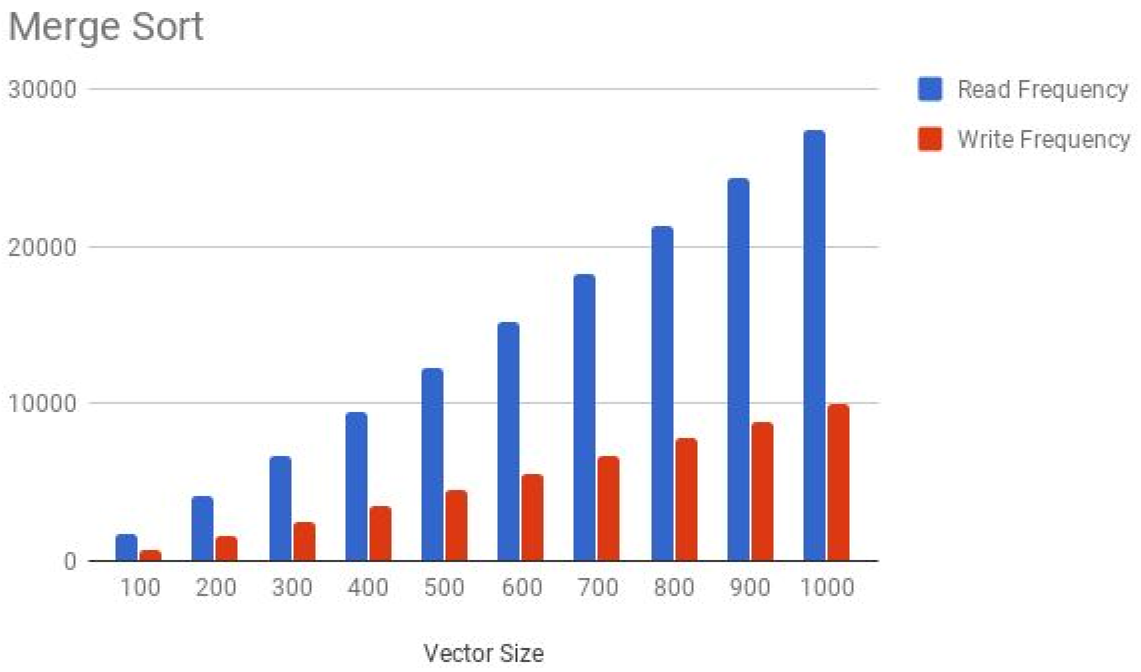
Code Source:

Dion, Lisa. “Main.cpp [Sorting Algorithms].” BlackBoard UVM, BlackBoard, 28 Feb. 2018,bb.uvm.edu/webapps/blackboard/execute/content/file?cmd=view&content_id=_2502870_1&course_id=_117709_1&framesetWrapped=true.

Time Complexity and Efficiency

The time complexity of merge sort is O(n log(n)) at worst-case. Below is a table of read and write frequencies recorded from sorting vectors of various sizes.

Vector Size	**Read Frequency**	**Write Frequency**
100	1756	672
200	4106	1544
300	6652	2488
400	9414	3488
500	12206	4488
600	15146	5576
700	18194	6676
800	21278	7776
900	24274	8876
1000	27402	9976



Heap Sort

Code Source:

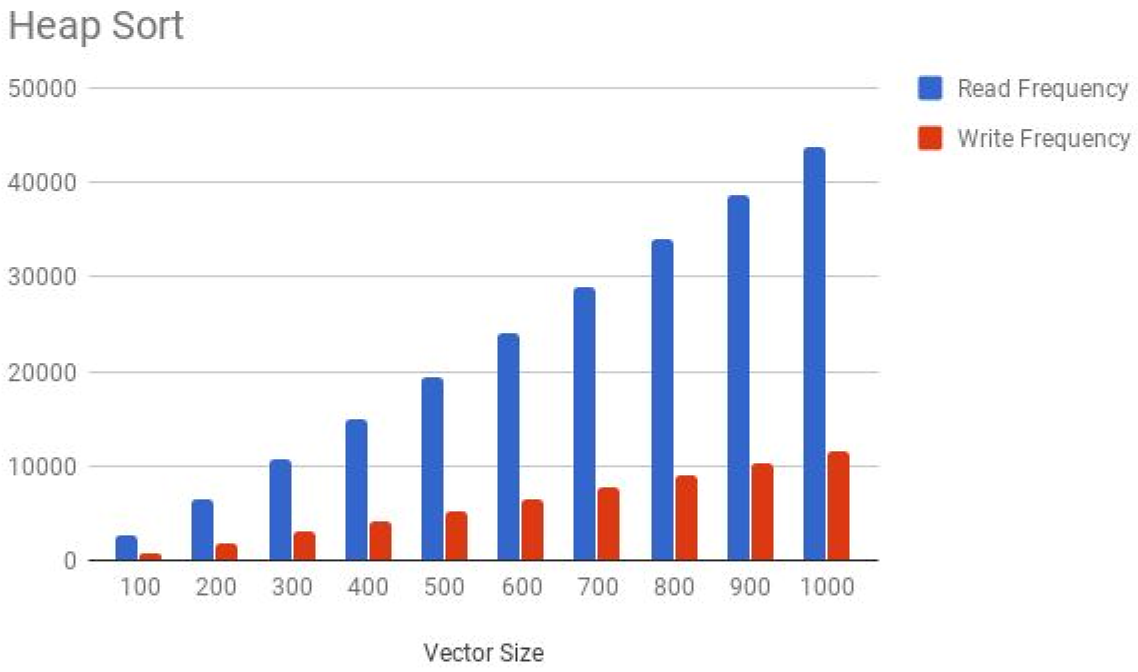
Weiss, Mark Allen. "Source Code for Data Structures and Algorithm Analysis in C++ (Fourth Edition)." CIU, 4th Edition, Ciu.edu, users.cs.fiu.edu/~weiss/dsaa_c++4/code/.

Time Complexity and Efficiency

The time complexity of heap sort is O(n log(n)) at worst-case. Below is a table of read and write frequencies recorded from sorting vectors of various sizes.

Vector Size	Read Frequency	Write Frequency
100	2726	823
200	6448	1855
300	10591	2968
400	14895	4114
500	19430	5279
600	24084	6491
700	28882	7727

800	33883	9022
900	38723	10264
1000	43675	11542



Two-Sort

Code Source:

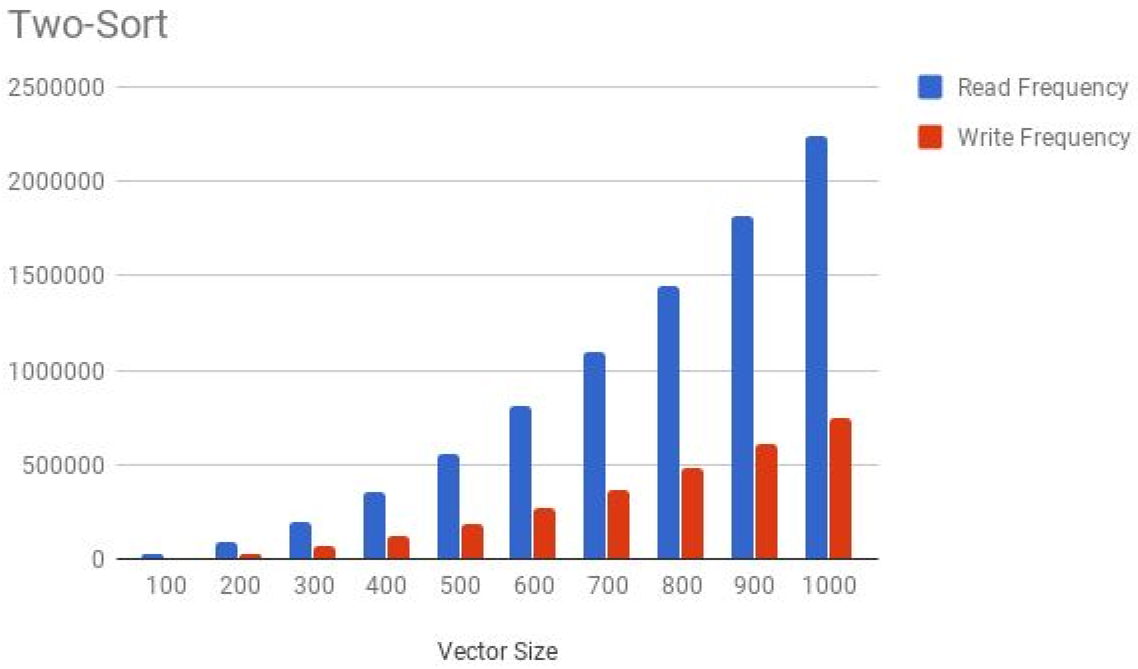
Dion, Lisa. “Main.cpp [Sorting Algorithms].” BlackBoard UVM, BlackBoard, 28 Feb. 2018,bb.uvm.edu/webapps/blackboard/execute/content/file?cmd=view&content_id=_2502870_1&course_id=_117709_1&framesetWrapped=true.

Time Complexity and Efficiency

Both insertion sort and bubble sort were used to sort by population, and by city name, in that order. The time complexity of insertion sort is O(n^2) at worst-case. The time complexity of bubble sort is O(n^2) at worst-case. Below is a table of read and write frequencies recorded from sorting vectors of various sizes.

Vector Size	**Read Frequency**	**Write Frequency**
100	22123	7348
200	88166	29110

300	195280	63611
400	353880	116750
500	560031	185893
600	810739	269864
700	1098775	363723
800	1441161	479100
900	1818015	604467
1000	2244153	745064



Conclusion

Of the sorting algorithms chosen above, it appears that merge sort reads and writes to the vector it is sorting far less than the other sorting algorithms. This would make it ideal for sorting a contacts list on a mobile app. When sorting a database of 20 million client files that are stored in a datacenter in the cloud, quicksort would be ideal because it is more efficient for much larger datasets.

Sincerely,
Max Peck